Prototyping the Recursive InterNet Architecture: The IRATI project approach

Sander Vrijders[1], Francesco Salvestrini[2], Eduard Grasa[3],
Miquel Tarzan[3], Leonardo Bergesio[3], Dimitri Staessens[1], Didier Colle[1]
[1]Ghent University - iMinds, INTEC, Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium
E-mail: firstname.lastname@intec.ugent.be
[2]neXtworks s.r.l., Pisa, Italy
E-mail: f.salvestrini@nextworks.it
[3]i2CAT Foundation, Jordi Girona, Barcelona, Spain
E-mail: firstname.lastname@i2cat.net

***Abstract*—In recent years, many new Internet architectures are being proposed to solve shortcomings in the current Internet. A lot of these new architectures merely extend the current TCP/IP architecture and hence do not solve the fundamental cause for these problems. The Recursive InterNet Architecture (RINA) is a true new network architecture, developed from scratch, building on experiences learned in the past. RINA prototyping efforts have been ongoing since 2010, but a prototype upon which a commercial RINA implementation can be built has not been developed yet. The goal of the IRATI reseach project is to develop and evaluate such a prototype in Linux/OS. This article focuses on the software design required to implement a network stack in Linux/OS. We motivate the placement of, and communication between, the different software components in either kernel or user space. A first open source prototype of the IRATI implementation of RINA will be available in June 2014 for researchers, developers and early adopters.**

I. Introduction

From the early days of telephony to nowadays, the telecommunications and computing industries have evolved significantly. However, we argue that they have been following separate paths, without achieving a full integration that can optimally support distributed computing; the paradigm shift from telephony to distributed applications is still not complete. Telecoms have been focusing on connecting devices, perpetuating the telephony model where devices and applications are the same. A look at the current Internet architecture shows many symptoms of this thinking [1]:

- The network routes data between interfaces of computers, as the public switched telephone network switched calls between phone terminals. However, it is not the source and destination interfaces that wish to communicate, but the distributed applications.
- Applications have no way of expressing their desired service characteristics to the network, other than choosing a reliable (TCP) or unreliable (UDP) type of transport. The network assumes that applications are homogeneous by providing only a single quality of service.
- The network has no notion of application names, and has to use a combination of the interface address and transport layer port number to identify different applications. In other words, the network uses information on "where" an application is located to identify "which" application this is. Every time the application changes its point of attachment it seems different to the network, greatly complicating multi-homing, mobility and security.

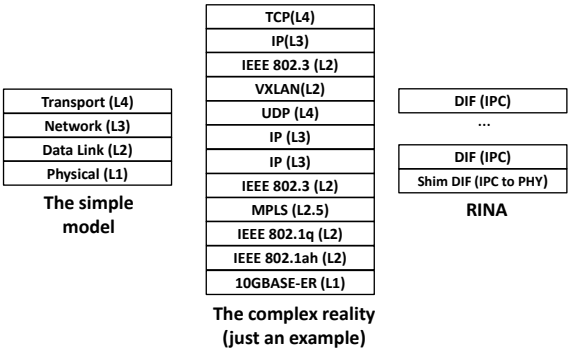

Fig. 1. Layers in Network Architectures.

Several attempts have been made to propose architectures that overcome the current Internet limitations. However, most of them still keep the same old assumptions, merely extending the architecture to handle more corner cases, at the price of complicating the general landscape more and more (see Figure 1).

A. The Recursive Internet Architecture

RINA, the Recursive InterNetwork Architecture [2] [3] [4], is a "back to basics" approach learning from the limitations of TCP/IP [5] and from the experience of other technologies in the past. With a clean-slate architecture, one focuses on solving the problem by offering a fundamental solution, in this case a global theory of networking, rather than hacking and patching the current technologies to ensure temporary operability.

RINA starts from the premise that *networking is Inter Process Communication (IPC) and IPC only*. Networking provides the means by which application processes on separate systems communicate, generalizing the model of local inter-process communication. Figure 2 shows a diagram of the

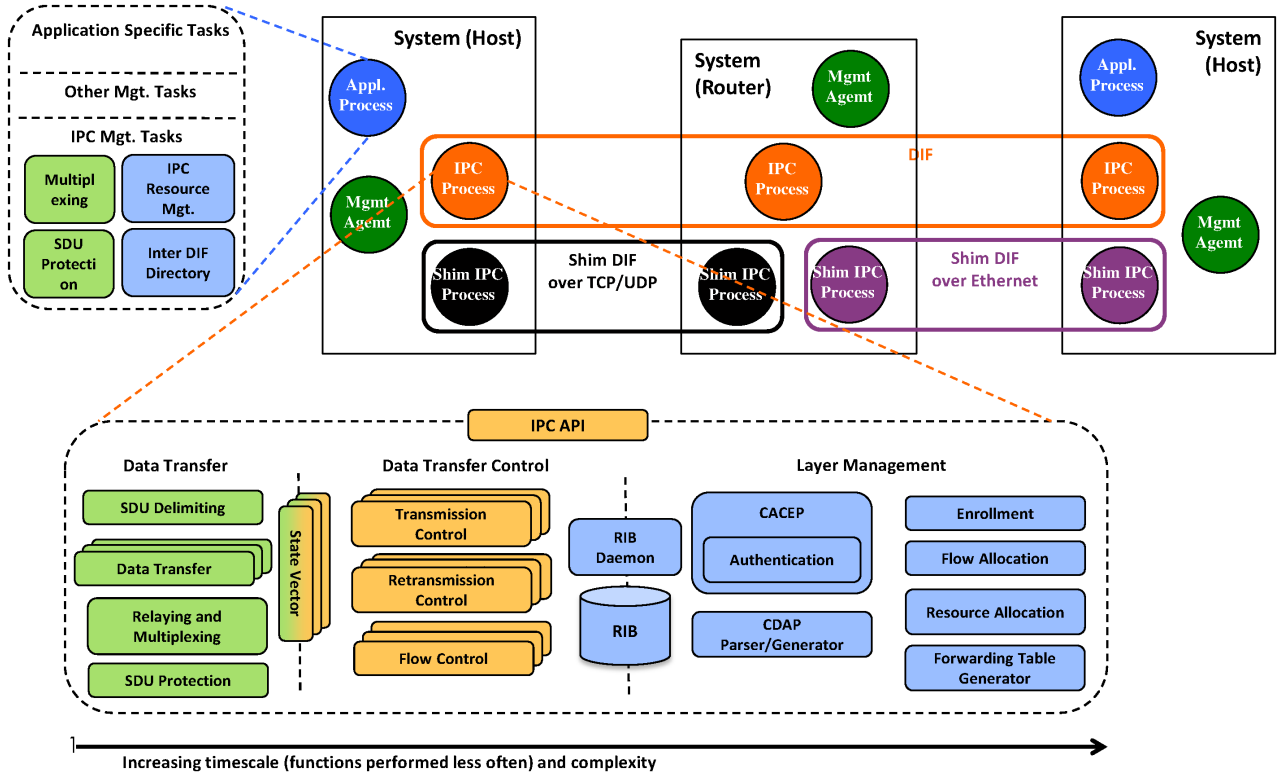

Fig. 2. The RINA Architecture Reference Model.

RINA architectural model. In contrast to the fixed, five-layer model of the Internet, where each layer provides a different function, RINAs architecture is based on a single type of layer, which is repeated in a recursive fashion as many times as required by the network designer. The layer is called a Distributed IPC Facility (DIF) which is a distributed application that provides Inter-Process Communication services over a given scope[1] to the distributed applications above (which can be other DIFs or regular applications).

A member of a DIF is called an IPC process. Normal IPC processes consist of elements dedicated to Data Transfer, Data Transfer Control or Layer Management. The elements are ordered in increasing complexity and frequency of use, with elements at the far left being used the most (per packet processing) but the least complex, and elements to the right being not often used, but very complex. All DIFs provide the same interface regardless of their rank with respect to other DIFs. This interface allows applications to request and manage IPC services by

- allowing an application to declare its reachability through one or more DIFs
- requesting the allocation of a communication flow to another application, specifying its name and the flow characteristics
- reading/writing data
- requesting the deallocation of the flow

[1]This scope can be a point-to-point link, a local area network, a metropolitan network, an internetwork, a virtual private network, etc.

Since all the layers provide the same functions, all layers have the same structure and components. However, not all the layers provide the same levels of service and operate over the same environment, therefore these components must provide means to adapt to a very heterogeneous environment. RINA follows the well-known Operating Systems (OS) design principle of separating mechanism (the invariant parts) and policy (the parts that can change) in each of its components. Therefore it is possible to customize the behavior of a DIF to optimally support a given set of applications or optimally operate on a certain environment by plugging in specific sets of policies instead of having to implement whole new protocols from scratch. For instance, there is Service Data Unit (SDU) protection, which allows protection of SDUs, depending on policy (no protection, encryption, ...).

B. Prototyping of the Recursive Internetwork Architecture

RINA prototyping efforts started in 2010, with three main objectives: debug and improve the initial draft RINA specifications, get experience on implementing RINA on different platforms and learn about the challenges and possibilities of overlaying it over TCP and UDP. There are currently three independent prototypes in different degrees of maturity, none of them aiming to be the basis of a platform that can be deployed in production; they all run in user-space:

- the Alba prototype, jointly developed by i2CAT and TSSG, entirely based on Java [6]

- the TRIA Network Systems prototype, a C implementation [7]
- the Boston University prototype, also written in Java. [8]

The IRATI project prototype [9] goes beyond the current state-of-the-art, since it is the first attempt to establish a source code base upon which commercial RINA implementations can be based. IRATI's implementation targets Linux/OS platforms, with components placed into both user and kernel spaces. This approach not only allows for performant implementations, but the placement of (at least) some components in kernel space is required to access low level functionalities, otherwise unaccessible. This allows to overlay RINA over different technologies (Ethernet, WiFi, USB, FireWire, etc.).

II. The manifold requirements of a clean-slate implementation

The IRATI project's overall high level software architecture spans the target OS as a whole, from the user space to the kernel space and identifies software components, their interactions and delineates their interfaces; the approach described in the project architectural deliverable (i.e. D2.1 which can be found at [9]) provides generic concepts that may help implementing the RINA model on different targets.

To accomplish the project goals, the focus of the IRATI software architecture is manifold and may be summarized as:

- optimally partitioning the components residing in both the user and kernel spaces
- defining their interfaces
- keeping both multitasking and user/kernel mode switches as low as possible, in order to avoid excessive performances degradation

The aforementioned requirements imply particular care on the design of the different components and the way they cooperate with each other. The identification of the operations executed on a regular basis is the groundwork for the analysis and consequent design activities. The functionalities that are most frequently used should suffer the lowest penalties and are described as residing in the "fast-path"; Protocol Data Units (PDUs) read/write operations are the easiest example. Operations that should not occur often (such as configuration or management ones) might suffer penalties without degradation, which is why they are addressed as residing in the "slow-paths".

By translating RINA's recursive approaches into iterative ones, e.g. to avoid abundant use of the stack, the positioning of components in either kernel or user space is easily obtained and fast-path operations among different processes in the same OS space can be obtained at the lowest context-switch cost. Placing the fast-path functionalities into kernel space prioritizes regularly executed tasks and reduces context switches as much as possible. On the other hand, the introduced partitioning, which places the software components in kernel or user space, means an increase in context switching. Hence, the best partitioning scheme implies an additional necessity, which is to have loose coupling between the two spaces' components and therefore functionalities spanning the two spaces should be avoided.

The aforementioned partitioning and coupling introduce problems that have been mildly relevant in traditional UNIX based systems, such as the intra-components communications which are now originated in both user and kernel spaces, compared to traditional IPC/RPC. Solving these problems optimally would incur the design of new user/kernel communications mechanisms, which are out of scope for the IRATI project. Since IRATIs goals are to implement a RINA stack with the current technologies available in its mainstream target kernels, such problems are addressed and solved with the minimum penalties on a per-case basis without requiring substantial modifications on the target systems while maximizing the software portability.

The IRATI implementation is split into three major layers: the application, the user-space (framework), which is mostly concerned with layer management tasks of the IPC Processes, and the kernel layer, which is mostly concerned with data transfer. The user-space framework is an Application Programming Interface (API) that exposes RINA functionalities to the applications and implements them with (traditional UNIX) daemons and (dynamic) libraries which in turn request services of the kernel on behalf of the applications. This common approach decouples the applications from the kernel particularities and enhances the portability of the user space software to other kernels with minor changes. The various dynamic libraries provide bindings to programs without constraining adopters to use only C and/or C++ languages: in order to suit the maximum audience, bindings to other languages (e.g. Java) are provided almost at no cost through the use of the Simplified Wrapper and Interface Generator (SWIG) [10].

III. High level software architecture

A. Components placement

The user-space framework consists of multiple components, which can be seen in Figure 3. The main functions and interactions between these components and the application can be summarized as:

- **Application Process**: An application using RINA services to communicate with other applications (residing on the same or on a different system).
- **IPC Process Daemons**: Each IPC Process daemon implements the layer management components of an IPC Process: the Resource Information Base (RIB), RIB Daemon, Enrollment, Flow Allocator, PDU Forwarding Table Generator and Resource Allocator components of the RINA reference model. The IPC Process is also an application process, and can request the allocation of N-1 flows to other IPC Processes in the system.
- **IPC Manager Daemon**: Manager of the RINA software in the system. It is in charge of creating, configuring and destroying the other components of the RINA software (both in user-space and the kernel). Interaction with the IPC Manager can be performed locally through a text

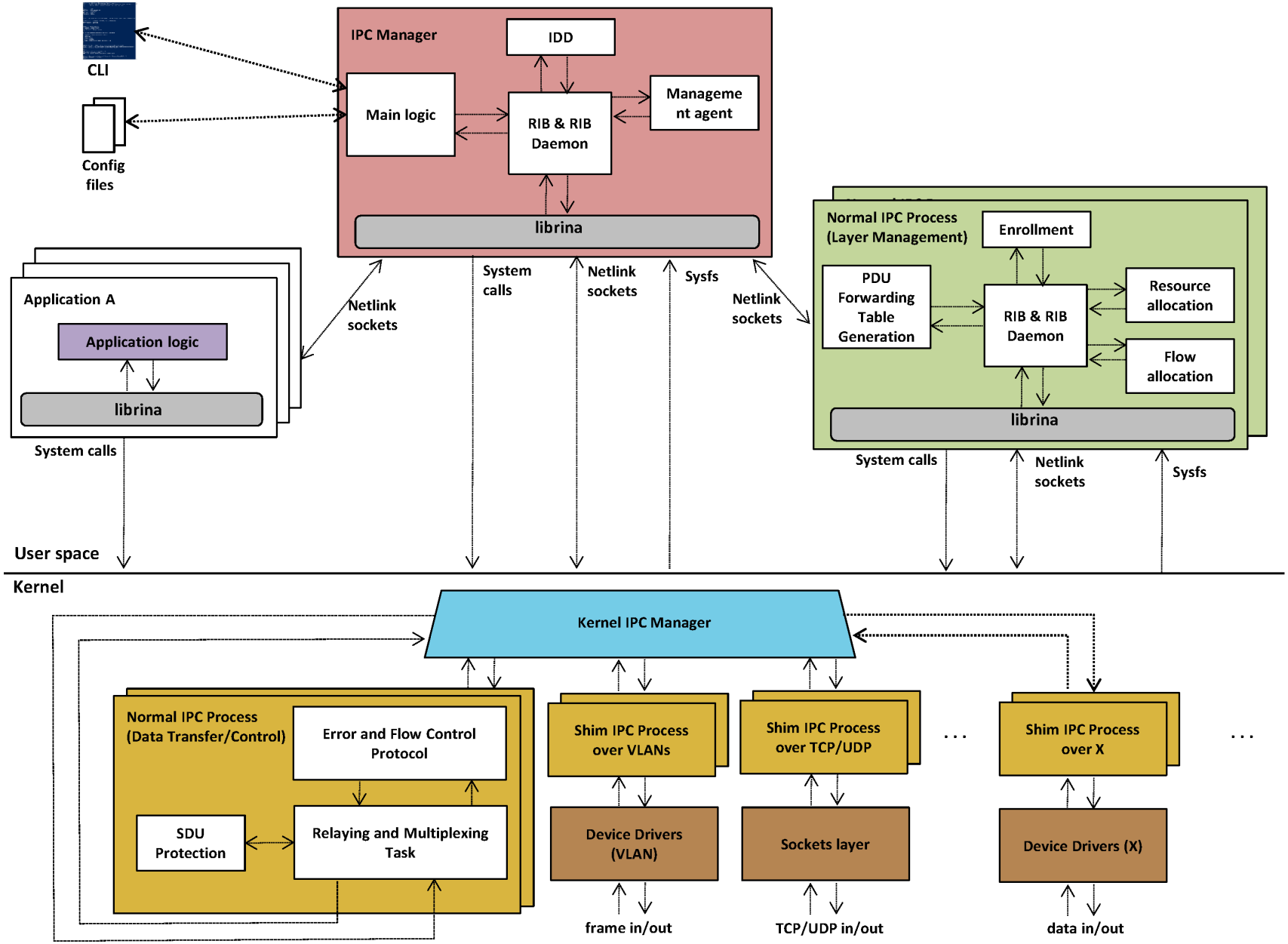

Fig. 3. The IRATI prototype components placement.

console providing a command-line interface, through configuration files, or remotely through a DIF Management System. The IPC Manager is also in charge of brokering application registration and flow allocation requests from applications, by redirecting these requests to the most appropriate IPC Process Daemon. Finally, the IPC Manager also implements the Inter-DIF Directory, which allows for the discovery of applications in DIFs not currently enrolled by the system. There is a single instance of an IPC Manager Daemon in each system.

The kernel layer also consists of multiple components. These components, their functions and interactions are:

- **Error and Flow Control Protocol (EFCP)**: Container of the different EFCP instances. EFCP is responsible for the actual data transfer. It is based on delta-t [11]. EFCP is split into the Data Transfer Protocol (DTP) and Data Transfer Control Protocol (DTCP) [12], loosely coupled through a state vector. There is an EFCP instance for each different connection in each IPC Process, which passes data to the Relaying and Multiplexing Task (RMT) in the outgoing direction, or to the Kernel IPC Manager in the incoming direction.
- **Relaying and multiplexing task (RMT)**: Container of the different RMT instances in the system. Each RMT instance (one per IPC Process) multiplexes the data from N EFCP connections to M underlying flows, and relays the data coming from underlying flows to EFCP connections or to other underlying flows with information from the forwarding table, generated by the PDU Forwarding Table. It passes data to the Kernel IPC Manager in the outgoing direction, and to EFCP in the incoming direction.
- **Service Data Unit (SDU) Protection**: Container of the different SDU Protection module instances in the system. At the finest granularity, there can be a different SDU Protection module instance for each distinct underlying flow. The SDU Protection component in the kernel is called by the RMT component to either protect data (outgoing direction) or expose data (incoming direction).
- **Shim IPC Process over Ethernet**: For migrating from current TCP/IP over Ethernet networks to RINA, the concept of a shim IPC Process was introduced. The task of a shim IPC Process is to put as small as possible veneer over a legacy protocol to allow a regular IPC process to use it unchanged. This component is the container of all the shim IPC Processes over Ethernet instances in the system. In the outgoing direction, this component passes data to the relevant device driver, and in the incoming direction, to the Kernel IPC Manager.
- **Shim IPC Process over TCP/UDP**: Container of all the shim IPC Process over TCP/UDP instances in the system. In the outgoing direction, this component acts as a gateway from RINA to TCP/UDP+IP, and in the incoming direction, to the Kernel IPC Manager.
- **Kernel IPC Manager**: Manages the lifetime (creation, destruction, monitoring) of the other component instances in the kernel, as well as its configuration. It also provides coordination at the boundary between the different IPC Processes. It passes data to EFCP or a shim IPC Process component in the outgoing direction, and to either the RMT or an application process at user space in the incoming direction.

B. Communications

Efficient communication mechanisms are a key requirement of the implementation. The RINA model itself implies IPC communication involving different processes and scopes. The modular design of the implementation creates a very interactive environment with the need of inter-module communication. A consequence of partitioning the components in user and kernel space is that communication is sometimes originated from the kernel to user-space. There are currently little mechanisms that allow communication initiated in the kernel.

For user-originated atomic actions, or belonging to fast-paths, the definition of a set of system calls provides an efficient and less resource consuming approach for synchronous user to kernel dialogues, since they don't require a socket for communication. When a socket is used for communication, the data first has to be serialized before it is send. Upon receiving, the data has to be deserialized. Sycalls are for instance used for Protocol Data Units (PDUs) read/write operations. Netlink sockets [13] also match many of our requirements (1:N, N:M, asynchronous, initiated communications by both-spaces) and make them suitable for the design and implementation of the IRATI framework. Netlink can support kernel to user-space communication due to its full-duplex channels, allowing the framework to warn the application about specific events. Netlink sockets and its Generic Netlink family are mainly used for notifications and batch-like communications for their asynchronous and multicasting capabilities (e.g. they are used for user-space component-to-component inter-communications and bidirectional communications between user and kernel spaces components). Finally, sysFS [14] allows to configure the different RINA components residing in kernel space, as well as obtaining metrics of these kernel components in user-space.

In user-space the communication framework abstracts the lower-level communication details. It is completely event-based in order to allow the different modules to react accordingly and update their state. This framework interacts with the different communications mechanisms used for user-space to user-space, user-space to kernel and kernel to user-space communication.

C. The user-space framework, applications and daemons

The user-space components and functionalities are provided by two separate software packages: librina and rinad. The librina package implements the aforementioned user-space components into the following libraries (see also Figure 3):

- **librina-common**: provides the common classes and utilities shared by all the libraries belonging to the librina framework.
- **librina-application**: provides the native RINA API, allowing applications to i) express their availability to be accessed through one or more DIFs (registration); ii) allocate and deallocate flows to destination applications (flow allocation/deallocation); iii) read and write data from/to allocated flows (in the form of SDUs) and iv) query the DIFs available in the system and their properties.
- **librina-cdap**: The Common Distributed Application Protocol (CDAP) allows applications to inter-communicate by performing operations in a well-defined syntax (create, delete, read, write, start or stop) on each application process Resource Information Base (RIB), which is the local representation of the view of the DIF in the IPC Process. Before any two application processes can exchange CDAP messages, they have to establish an application connection in order to i) authenticate each other if required and ii) agree on a version of the syntax and specific encoding of the application protocol to be used for the communication. The application connection procedure in RINA is defined by the Common Application Connection Establishment Phase (CACEP) [6], which provides hooks to plug-in authentication modules. librina-cdap provides an implementation of both the CACEP and CDAP state machines.
- **librina-faux-sockets**: supports legacy applications, albeit under-performant, without requiring a migration to the native RINA API. This library mimics the sockets API, mapping most of the sockets operations to operations on flows using the native RINA API. This way legacy application traffic can be carried over DIFs, without applications being aware of that (at the cost of not enjoying all the features of the native API).
- **librina-ipc-manager**: provides the functionalities used by the IPC Manager to perform IPC Process creation, deletion and configuration, to serve requests by application processes, and manage the Inter-DIF Directory. It abstracts the details that allow the IPC Manager to communicate with the IPC Process daemons and application processes in user space (using Netlink) and the RINA components in the kernel (using system calls, Netlink sockets and sysfs).
- **librina-ipc-process**: encapsulates the operations that allow the IPC Process daemon to serve the requests of application processes, to allow configuration by the IPC Manager and to configure the IPC Process components residing in the kernel. (e.g. creation/modification/deletion of EFCP instances, updates to the PDU forwarding table or the modification of the kernel SDU Protection policies). IPC Manager configuration requests involve the assignment of the IPC Process to a DIF, the registration to one or more N-1 IPC Processes or the triggering of an enrollment operation.
- **librina-sdu-protection**: This library provides a combination of procedures to detect bit errors, encrypt/decrypt SDUs and other features.

The rinad package contains the two daemons of the RINA framework in user-space: the IPC Manager daemon and the IPC Process daemon. Both daemons make use of the librina libraries to accomplish their tasks.

Applications can be written using the aforementioned framework by simply linking these libraries and adopting their APIs. By following this approach, portability problems among different systems and issues caused by the underlying kernel level API/ABI changes are reduced to the minimum.

The librina package is not limited to provide C bindings only, as the majority of libraries available. It currently provides bindings to C, C++ and Java languages. More languages, e.g. Python and Ruby, are expected to be supported in the future. Part of these bindings (i.e. Java) are automatically generated using SWIG, which is used to parse the librina C/C++ interfaces and automatically generate the "glue code" required for the target languages to use the wrapped interfaces. Due to the approach taken with regard to wrapping, future development only takes a small amount of effort compared to the handwriting of bindings for all the targeted languages. By extending this procedure to other languages, the reuse of the project's solutions is encouraged and is expected to be a real advantage, compared to the efforts spent implementing them.

D. The kernel-space framework and components

The kernel space software has different requirements and targets compared to the user-space, e.g. a well defined set of programming languages (i.e. only assembly and C), different memory and resources models, different synchronization and concurrency techniques, development patterns and, last but not least, a generally harsher environment compared to the user-space one.

In order to overcome all the possible limitations of the environment, reduce the problems that may be caused by incrementally introducing features during the whole project lifetime and keeping code refactoring at the minimum, we followed an Object Oriented (OO) approach throughout all our kernel-level components. OOP approaches applied to C development are not new however, e.g. the early C++ compilers prototypes were almost preprocessors producing all the C required boilerplates, and a lot of literature is easily available in the public domain [15]. It is worth noticing however that these approaches and techniques usually deal with C in user-space and must be refactored opportunely to be applied into kernel space. Part of these OOP "techniques" are already in use in the Linux kernel (and we applied them in our implementation accordingly) while the remaining ones have been introduced during the development phase where needed. To simply introduce some of them, without the intention of discussing them in detail:

- Information hiding: Implemented through the use of forward definitions of the object's type in its header file while the complete definition is embedded into the corresponding compilation module

- Constructors and destructors: Implemented through the use of functions mimicking (C++) constructors and destructors.
- Methods, polymorphism and inheritance: Implemented through the use of function pointers defined into the corresponding object type. By opportunely mangling these pointers and enforcing common naming constraints, polymorphism and inheritance can be easily obtained

By applying such techniques, the internal implementation of a stack component can be hidden and its interface "exported" as a set of function pointers and an opaque data pointer (pointing to the component's internal state). That interface can be dynamically bound into other components by storing these function pointers and data pointer into the targets, received through the use of an accessory function. The opposite operation, the unbinding, is similar.

With the generalized adoption of the previously discussed approaches throughout the stack and with the kernel provided runtime dynamic linking features (i.e. module loading and unloading), the implementation of dynamical embedding and removal of features into the stack at runtime is possible. Such characteristic avoids the need to change the core of the stack each time the internal implementation of a component changes, which is especially important for the shim IPC processes since they strongly depend on the technology underneath (e.g. Ethernet, WiFi) and thus their implementation vary widely. This way, new technologies can easily be made available to the RINA stack.

The dynamic components binding is primarily used inside the Kernel IPC Manager (KIPCM), which is also in charge of providing the entry-points of the RINA syscalls. The KIPCM exposes its hooks to the syscalls layer which in turn are bounded to the lower parts of the stack, the shims. A reverse-chain of initializations and bindings (i.e. the shims IPC Processes initialized and bounded to the core of the stack which in turn is finally bounded to the RINA syscalls) provides the most viable way to implement the dynamic plugging of the shims IPCs at boot-time, as well as during runtime.

The overall OO approach adopted shows its potential into the IPC processes layer: the KIPCM provides the same binding and usage APIs for both the normal and the shim IPC Processes, regardless of their "exact" type. This way all the functionalities they provide are transparent (and homogeneous) to the upper layers, even though their inner workings may vary greatly. For instance, the "regular" IPC Processes have EFCP and RMT instances while the shims are completely missing them, but the user-space components working in the fast-paths are unaware of such differences.

IV. Future work

The official IRATI project implementation roadmap is sliced into three phases, providing incremental features on top of each other, towards the implementation of an open source, full fledged, RINA stack. This roadmap can be summarized as follows:

- **1st phase (restricted prototype, Nov. 2013)**: This prototype targets a first implementation of all the components with limited functionalities which will allow the creation of DIFs over Ethernet. This will provide levels of service similar to TCP/IP and UDP/IP, with the added features of supporting authentication when joining the DIF, and inherent support for mobility and multi-homing, as is the case for RINA.
- **2nd phase (open source prototype, Jun. 2014)**: This prototype will be an evolution of the first one, fixing known bugs and introducing functionalities not present in the 1st prototype such as supporting levels of service different to those of TCP and UDP thanks to the development of different resource allocation policies and the implementation of SDUs protection mechanisms.
- **3rd phase (open source prototype Dec. 2014)**: As in the 2nd phase, this prototype will be fixing bugs and introducing the remaining functionalities towards a feature-complete state. Compared to the other phases, the focus is on the interoperability with the non-IRATI prototype(s) and the completion of the eventual open issues that could not have been addressed before.

Upon the first public release (2nd phase prototype), the IRATI project will start promoting an open source community to further develop and enhance the public prototype. Apart from the aforementioned official releases, development ones will be rolled out more often and the repository will be shared with the community.

V. Conclusion

RINA, the Recursive InterNet Architecture is a "back-to-basics" approach on network architectures. We presented the design decisions made in the IRATI prototype of RINA, the first open source code base upon which commercial RINA implementations can be based. We outlined and motivated the placement of the components in user and kernel space and described their functionality and the interactions between them. The first open source prototype will be available in June 2014, which can be a starting point for developers and/or be used to to experiment with.

Acknowledgment

This work is partly funded by the European Commission through the IRATI project (Grant 317814), part of the Future Internet Research and Experimentation (FIRE) objective of the Seventh Framework Programme (FP7).

References

[1] A. McKenzie, "INWG and the conception of the Internet: An eyewitness account," *Annals of the History of Computing, IEEE*, vol. 33, no. 1, pp. 66–71, 2011.

[2] J. Day, *Patterns in network architecture: a return to fundamentals*. Prentice Hall, 2008.

[3] V. Ishakian, J. Akinwumi, and I. Matta, "On the Cost of Supporting Multihoming and Mobility," Boston University Computer Science Department, Tech. Rep., 2009.

[4] J. Day, E. Trouva, E. Grasa, P. Phelan, M. P. de Leon, S. Bunch, I. Matta, L. T. Chitkushev, and L. Pouzin, “Bounding the router table size in an ISP network using RINA,” in *Network of the Future (NOF), 2011 International Conference on the*. IEEE, 2011, pp. 57–61.
[5] J. Day, “How in the heck do you lose a layer!?” in *Network of the Future (NOF), 2011 International Conference on the*. IEEE, 2011, pp. 135–143.
[6] E. Grasa, E. Trouva, S. Bunch, P. DeWolf, and J. Day, “Developing a RINA prototype over UDP/IP using TINOS,” in *Proceedings of the 7th International Conference on Future Internet Technologies*, 2012, pp. 31–36.
[7] (2013, Nov.) TRIA Network Systems, LLC. [Online]. Available: http://www.trianetworksystems.com/
[8] Y. Wang, F. Esposito, I. Matta, and J. Day, “Recursive InterNetworking Architecture (RINA) Boston University Prototype Programming Manual (version 1.0).”
[9] (2013, Nov.) The IRATI website. [Online]. Available: http://www.irati.eu
[10] D. M. Beazley *et al.*, “SWIG: An easy to use tool for integrating scripting languages with C and C++,” in *Proceedings of the 4th USENIX Tcl/Tk workshop*, 1996, pp. 129–139.
[11] R. W. Watson, “Timer-based mechanisms in reliable transport protocol connection management,” *Computer Networks (1976)*, vol. 5, no. 1, pp. 47–56, 1981.
[12] G. Gursun, I. Matta, and K. Mattar, “Revisiting a soft-state approach to managing reliable transport connections,” in *Proceedings of the Eighth International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*, 2010.
[13] P. Neira-Ayuso, R. M. Gasca, and L. Lefevre, “Communicating between the kernel and user-space in Linux using Netlink sockets,” *Software: Practice and Experience*, vol. 40, no. 9, pp. 797–810, 2010.
[14] P. Mochel, “The sysfs filesystem,” in *Linux Symposium*, 2005, p. 313.
[15] A.-T. Schreiner, “Object oriented programming with ANSI-C,” 1993.